Algorithm: Lab4 (By Sujiv Shrestha ID:610145)

Problem 1.

1. Determine whether InsertionSort, BubbleSort, SelectionSort from lesson 3 are *stable* sorting algorithms, and in each case, explain your answer.
Answer:

InsertionSort: Insertion sort is stable sorting algorithm because it starts with comparing the first element of array and shifts the elements to its left only when that element is greater than the compared element and continues so forth. This way it stops shifting elements when it finds duplicates and hence the sorting is stable.

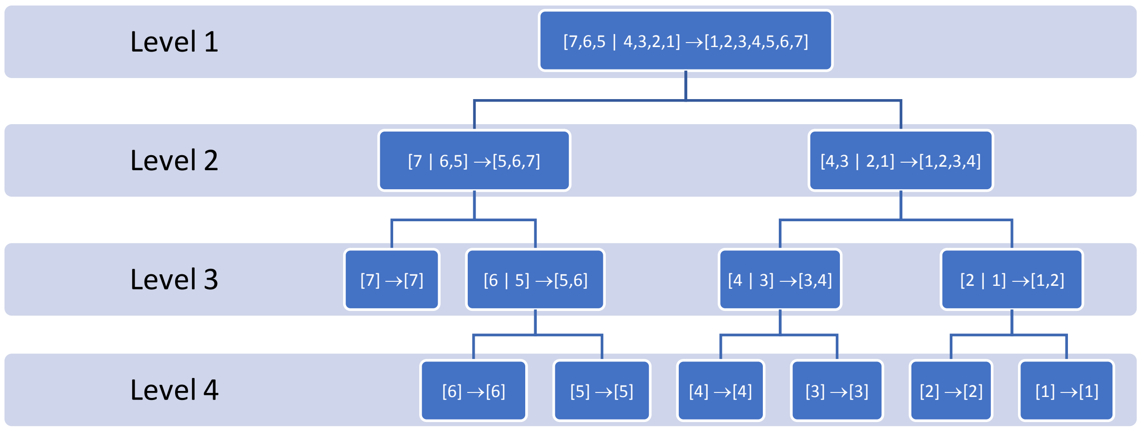
BubbleSort: Bubble sort is also stable as implemented in the test code from lesson 3 because it involves only swapping of inversion-pairs. When there are duplicates in any pair the swapping is not performed.

SelectionSort: Selection sort as implemented in lesson 3 is not stable because the the sorting involves finding the minimum of selection scope of array and swaping that minimum with the left most element of that selection. Doing so there is a possibility that the original order of the duplicates are not maintained in the final sorted array.

Problem2

2. Perform the MergeSort algorithm by hand on the array [7, 6, 5, 4, 3, 2, 1]. Show all steps.

```
Algorithm mergeSort(S)
    Input sequence S with n integers
    Output sequence S sorted
    if S.size() > 1 then
        (S1, S2) ← partition(S, n/2)
        mergeSort(S1)
        mergeSort(S2)
        S← merge(S1, S2)
    return S
```



Problem3

3. Sometimes MergeSort is supplemented with a secondary sorting routine (typically, InsertionSort is used) in the following way: During the recursion in MergeSort, the size of the array being sorted becomes smaller and smaller. To create a hybrid sorting routine, when a recursive call requires the algorithm to process an array with 20 or fewer elements, give this array to InsertionSort and patch in the result after it has finished. Call this hybrid algorithm MergeSortPlus.

A. Express the steps of MergeSortPlus in the pseudo-code language we are using in class.

```
Algorithm mergeSortPlus(S)
      Input sequence S with n integers
      Output sequence S sorted
      if S.size() > 1 then
            if S.size() < 21 then
                  S←insertionSort(S)
            else
                  (S1, S2) ← partition(S, n/2)
                  mergeSort(S1)
                  mergeSort(S2)
                  S← merge(S1, S2)
      return S
```

B. Write the Java code for MergeSortPlus (use the implementation of MergeSort provided in the lab folder)

Solution=>

```
void mergeSort(int[] tempStorage, int lower, int upper) {
    if(lower==upper){
          return;
    }
    else if(upper-lower<=20) {
          insertionSort(lower,upper);
    }
    else {
          int mid = (lower+upper)/2;
          mergeSort(tempStorage,lower,mid);  //sort left half
          mergeSort(tempStorage,mid+1, upper); //sort right half
          merge(tempStorage,lower,mid+1,upper); //merge them
    }
}
private void insertionSort(int lower, int upper) {
    if(lower >= upper) {
          return ;
    }
    int j = 0;
    for(int i = lower+1; i < upper+1; ++i) {
          int tmp = theArray[i];          //theArray is input array
          j=i;
          while(j>lower && tmp < theArray[j-1]){
                theArray[j] = theArray[j-1];
                j--;
          }
```

```
                theArray[j]=tmp;
            }
        }
```

C. Run tests to compare running times of MergeSort and MergeSortPlus. Which one runs faster? Explain how you tested and whether you feel your results are conclusive.

Answer: The result after implementing MergeSortPlus and testing with other sorting algorithm gave following results:

```
2 ms -> BubbleSort2
2 ms -> BubbleSort1
28 ms -> MergeSortPlus
39 ms -> MergeSort
292 ms -> InsertionSort
480 ms -> SelectionSort
1605 ms -> BubbleSort
```

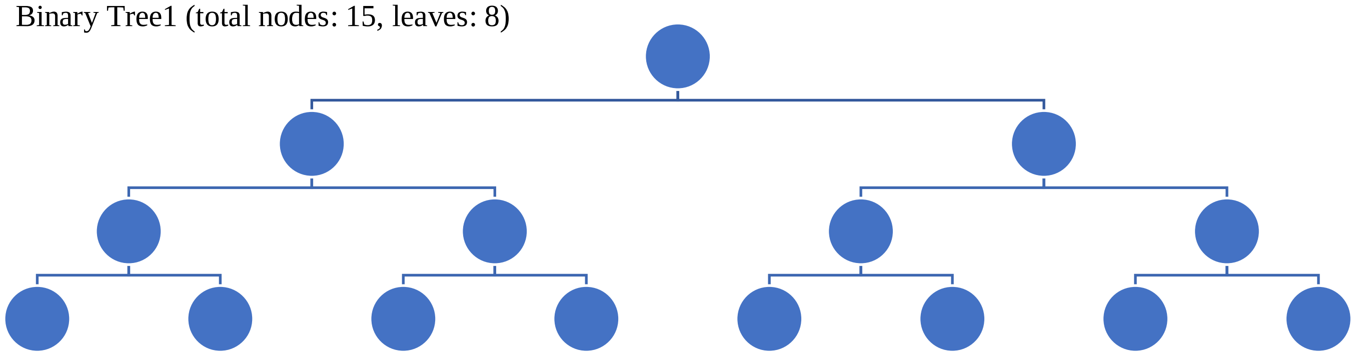
The result showed that MegerSortPlus runs little faster than MergerSort. The result is as predictable because when we limit the size of input array to lower values like 20 for all sorting algorithm then we find that insertionSort sorts the array faster than other algorithm. So, ones the sub array size reaches to 20 or less it is wise to use insertion sort than merge sort itself.

Problem4

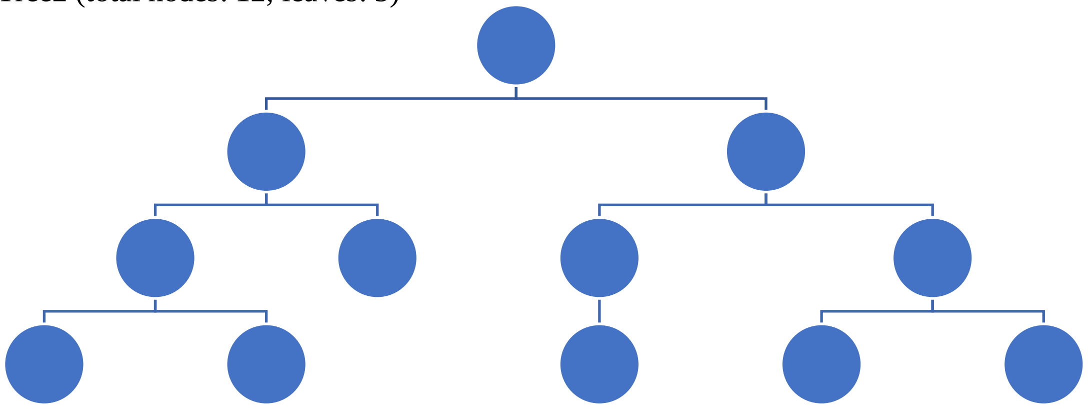
4. *Binary Trees*. A *binary tree* is a tree in which every node has at most two children.

a. Write out 4 different binary trees, each having height = 3 – make sure that no two of your trees have the same number of nodes. (There is no need to give labels to the nodes.)

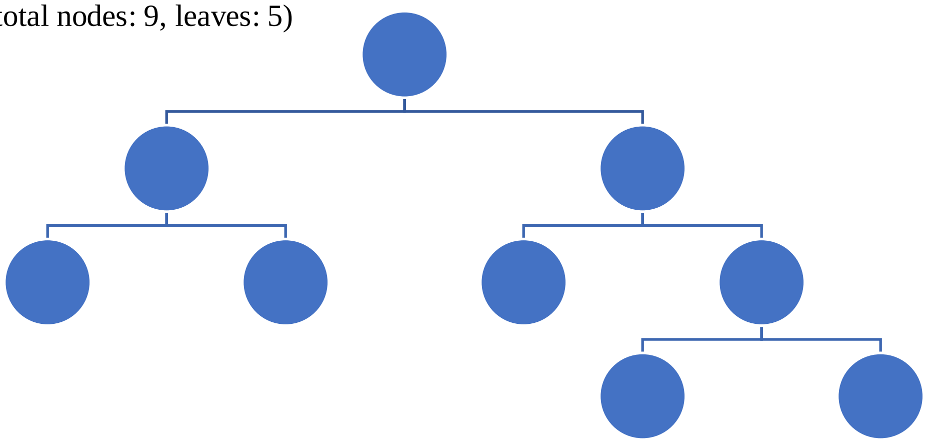
Binary Tree1 (total nodes: 15, leaves: 8)



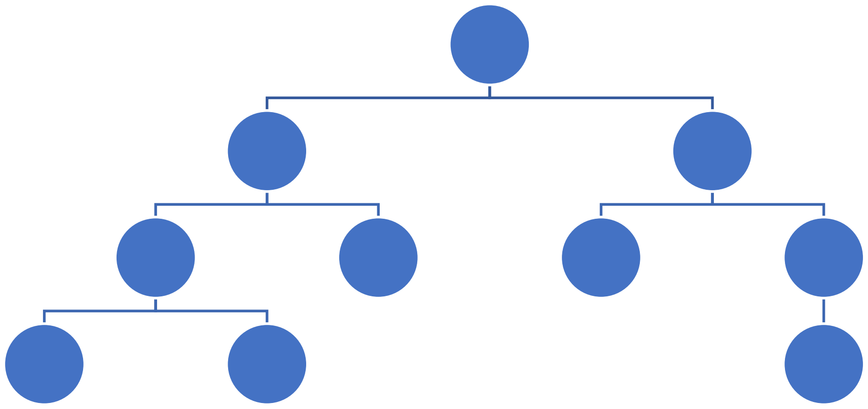
Binary Tree2 (total nodes: 12, leaves: 5)



Binary Tree3 (total nodes: 9, leaves: 5)



Binary Tree4 (total nodes: 10, leaves: 5)



b. Examine the trees you have drawn and decide whether the following statement is true or false:
Every binary tree of height 3 has at most 23=8 leaves.
Answer: True

c. Based on your answer to b, what do you think is true in general about the number of leaves of a binary tree of height *n*?
Answer: Number of leaves of a binary tree $\leq 2^{\text{height of binary tree}}$

Problem5

5. Solve the following problem with a recursive algorithm: Given a list with n elements, put the elements of the list in reverse order. Compute the running time of your algorithm (hint: count self-calls).

```
Algorithm reverse(S)
    Input sequence S with n integers
    Output sequence S in reverse order
    if S.size() > 1 then
        (S1, S2) ← partition(S, n/2)
        reverse(S1)
        reverse(S2)
        S← combine(S2, S1)
    return S
```

Java Implementation:

```
	private static int[] reverse(int[] arr) {
		if(arr.length>1) {
			int[] a = reverse(Arrays.copyOfRange(arr, 0, arr.length/2));
			int[] b = reverse(Arrays.copyOfRange(arr, arr.length/2, arr.length));
			arr = combine(b,a);
		}
		return arr;
	}

	private static int[] combine(int[] x, int[] y) {
		int[] z = new int[x.length+y.length];
		int c = 0;
		for(int i:x) {
			z[c++] = i;
		}
		for(int j:y) {
			z[c++]=j;
		}
		return z;
	}
```